Dynamic Prefiltering for Massive-Scale Equi- and Geospatial Joins

Jason Arnold
Knut Stolze
Ellis Saupe
{jarnold,stolze,esaupe}@ocient.com
Ocient, Inc.
Chicago, IL, USA

Abstract

Modern hyperscale analytical data warehouses execute joins over trillions of rows, where performance is dominated by I/O, data shuffling, and CPU-intensive join processing. While runtime filtering is common for simple equi-joins, existing techniques are typically limited to simple predicates and are loosely integrated with the underlying execution engine, often resulting in suboptimal performance for complex workloads such as geospatial analytics.

We present a unified framework for dynamic join prefiltering in the Ocient Hyperscale Data Warehouse, a parallel system built on a Compute Adjacent Storage Architecture (CASA) and using a push-based execution model. The construction and application of prefilters are integrated with Ocient's fine-grained scheduler. Our framework generates prefilters on the build side of a join at runtime and propagates them directly to storage-adjacent I/O operators on the probe side. For equi-joins, we introduce a graceful degradation that transitions from exact key lists to probabilistic Bloom filters. For complex geospatial joins over geodetic data, we extend the same mechanism with lists of bounding boxes and a novel tiling-based query rewrite: spatial joins are transformed into parallel hash joins on dynamically generated tile identifiers, where passing rows are deduplicated via an efficient owning-tile resolution scheme. To address the limitations of list-based spatial filtering, we also present the design of "GeoFilter," a novel probabilistic quadtree structure for adaptive spatial pruning.

This integration of prefiltering with query scheduling, execution, and I/O reduces data movement and computational work at petabyte scale. Our evaluation demonstrates that dynamic prefiltering improves standard analytical benchmark performance by over 20% and reduces latency for selective geospatial joins by over 97%.

Keywords

Dynamic join prefiltering, Runtime filtering, Geospatial joins, Hyperscale data warehouse, Parallel database systems

1 Introduction

Processing joins on hyperscale datasets, often comprising trillions of rows, is a key challenge in modern analytical data warehouses. Storing all data in memory is not always feasible [10, 24]. The cost of these operations is typically dominated by I/O, network data transfer during shuffles, and CPU load within the join operators themselves. While traditional query optimizers rely on static statistics to create efficient plans, these estimates can be imprecise, leading to suboptimal performance [15, 16]. This is particularly true for complex, non-equi-joins, such as those found in geospatial analytics, where data skew and predicate complexity are common.

The Ocient Hyperscale Data Warehouse (Ocient) is a Massively Parallel Processing (MPP) system designed for large-scale analytics, built upon a Compute Adjacent Storage Architecture® (CASA) [26]. In CASA, high-throughput NVMe storage is directly attached to dedicated CPU cores, enabling data-proximate filtering and processing to minimize data movement. Query execution within Ocient follows a push-based model, where each operator is an independent entity that performs some operation on the data it receives. Operators communicate through small, efficient single-producer, single-consumer (SPSC) queues.

Along with providing fine-grained parallelism, this architecture also offers specific opportunities for dynamic runtime query optimization.

In this paper, we present a unified framework for dynamic join prefiltering implemented in Ocient. This framework leverages our join scheduling strategies to generate filters from the smaller (build) side of a join *while it is executing*. These filters are then propagated to operators that can apply them—most notably I/O operators processing the large (probe) side—enabling them to prune a large volume of non-qualifying data directly at the source (cf. Figure 1).

We detail two primary applications of this framework:

Equi-Join Prefiltering: A graceful degradation strategy that begins with an exact list of values in the join key columns, transitions to a probabilistic Bloom filter (BF) as data volume scales, and is finally disabled if selectivity is lost.

Geospatial Join Prefiltering: An analogous system using lists of bounding boxes to accelerate intersect-style joins, integrated with a query rewrite strategy that transforms complex spatial joins into high-performance tiled hash joins.

Our core contribution is the design and implementation of a holistic prefiltering system, integrated with the query execution and scheduling model. This leads to a reduction of data movement and computation for both common and complex join patterns at petabyte scale.

The remainder of this paper is organized as follows. Section 2 describes the architectural foundations of Ocient, focusing on the push-based execution engine and join scheduling strategies that create the opportunity for dynamic prefiltering. Section 3 presents our prefiltering framework for equi-joins, including filter generation, propagation, and graceful degradation. Section 4 extends this framework to geospatial workloads, detailing bounding-box-based prefiltering and the tiled execution model for large-scale spatial joins. An experimental evaluation of these

EDBT '26, Tampere (Finland)

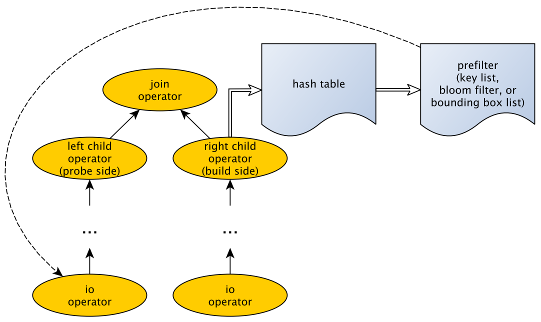

join operator
hash table
prefilter (key list, bloom filter, or bounding box list)
left child operator (probe side)
right child operator (build side)
...
io operator
io operator

Figure 1: Query plan for a join with dynamic prefiltering

techniques on realistic hyperscale workloads is given in Section 5. Section 6 discusses current limitations and our future ideas for probabilistic geospatial prefiltering. Section 7 situates our approach within the broader literature on runtime filtering and spatial query processing, and Section 8 concludes the paper.

2 Architectural Foundations

The effectiveness of our dynamic prefiltering strategy is a direct consequence of Ocient's core execution model. Ocient employs a push-based, multi-threaded execution engine where operators run as independent tasks.

2.1 Push-Based Execution and Parallelism

A query plan in Ocient is instantiated across tens of servers and hundreds of CPU cores per server. Each operator in the plan is instantiated and runs in parallel on every available thread. Data flows from child operators to parent operators through small, bounded SPSC queues (cf. Figure 2). An operator instance becomes runnable when it has input data available in its input queue(s), its required resources (e.g., memory) are met, and there is space available in its output queue(s). This model creates a highly concurrent environment where different parts of a query plan can execute independently. Certain operators, such as the shuffling of data within a node, act as multiplexers, utilizing an N^2 mesh of SPSC queues to communicate between all threads on a node.

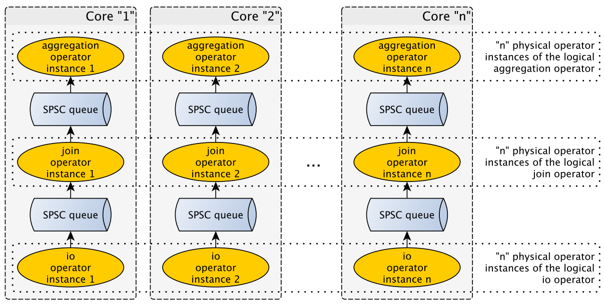

Core "1"
Core "2"
Core "n"
aggregation operator instance 1
aggregation operator instance 2
aggregation operator instance n
"n" physical operator instances of the logical aggregation operator
SPSC queue
join operator instance 1
join operator instance 2
...
join operator instance n
"n" physical operator instances of the logical join operator
io operator instance 1
io operator instance 2
io operator instance n
"n" physical operator instances of the logical io operator

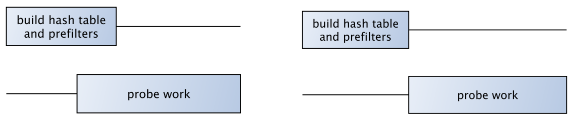
Figure 2: Inter-operator communication via SPSC queues

2.2 Join Scheduling Strategies

The query scheduler manages execution of concurrent branches in a plan, such as the left and right inputs to a join. This scheduling is key to creating the opportunity for prefiltering. We employ two primary strategies (cf. Figure 3):

Weak Right (`WEAK_RIGHT`): Used for inner and semi-joins, this strategy prioritizes executing the right (build) side of the join to construct the hash table. However, if all operators on the right branch are blocked (e.g., waiting for I/O or output queues to clear), the scheduler is permitted to execute runnable operators on the left (probe) branch. This ensures maximum resource utilization but implies that the probe side may begin processing data *before* the build side has completed.

Strict Right (`RIGHT`): Used for geospatial joins, this strategy is more rigid. It mandates that the entire right branch must complete execution before any data is processed on the left branch. We found this necessary for geospatial workloads, where the performance penalty of scanning even a small fraction of the probe side without a bounding box filter was unacceptably high and caused queries to slow down.


build hash table and prefilters
probe work
build hash table and prefilters
probe work

(a) Weak Right **(b) Strict Right**

Figure 3: Scheduling Strategies

These scheduling models create a critical time window: after the right side begins executing but before the left side has been fully processed, information gathered from the right side can be used to optimize the left. This timing allows for dynamic prefiltering to be employed at runtime.

3 Dynamic Prefiltering for Equi-Joins

During execution of a standard hash join, the right side is scanned to build a hash table in memory.[1] Our prefiltering mechanism piggybacks on this process.

3.1 Filter Generation and Propagation

After the hash table is built on a thread, the keys of the hash table are scanned to determine which kind of prefilter to construct:

Key List: A simple list containing the unique values from the join key columns encountered on the right side.

Bloom filter: A probabilistic data structure representing the same set of values from the join key columns [2]. Our implementation is backed by huge pages and uses AVX2 instructions for high performance.

In the case of a multi-join, these thread-local filters may be regenerated or updated each time a build-side child has finished adding all of its rows to the hash table. These thread-local filters are initially aggregated on each node and then combined across the cluster to form a single, global view of the values from the join key columns from the entire right-side input. Depending on the way data is partitioned among operator instances on the left branch of the join, it may be possible to apply thread-local or node-local filters even before the global filter is complete. For

[1]In Ocient, the build side of a hash join is always the right side. In case of a multi-join, data from multiple children is built into a combined hash table, which is later probed by the leftmost join child. `WEAK_RIGHT` or (strict) `RIGHT` scheduling applies sequentially to multi-join branches; a 3-child multi-join attempts to complete its rightmost build child, then the middle build child, then the probe child.

example, Figure 4 shows a *multiplexer* operator, which repartitions the data within a node but not across node boundaries. All thread-local filters generated from each join operator instance on the node must be unioned to form a node-local filter. This node-local filter is passed on to the *select* operator for earlier filtering. Similarly, a cross-node shuffle operator may receive a copy of each node-local filter and apply this more selective filter directly to its associated partition. All node-local filters are combined to form the global filter by the *prefilter manager*. The prefilter manager is takes care of passing node-local as well as global filters directly to the operators that can benefit from applying them.

This process enforces a strict completeness guarantee: if any single node discards its local filter (due to saturation or size limits), the global filter construction is aborted, and prefiltering is disabled for that query to prevent incorrect partial filtering. Once a global filter has been materialized, it is sent directly to all of the operators that can benefit from applying it. The most important recipients are the I/O operator instances reading the tables for the left (probe) side of the join.

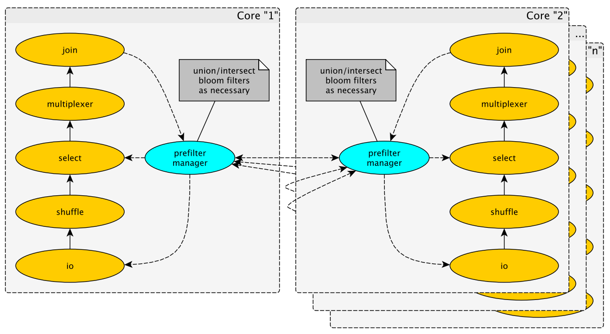


Figure 4: Communication of prefilters on a single core and across cores/nodes

Prefilters are generated once the hash table construction on the right side is complete on a given thread. Building the complete hash table first removes the need to deduplicate keys while building the prefilter. It allows us to decide upfront which kind of prefilter to build: key list and BF, BF only, or no prefiltering entirely (cf. Section 3.2). Scanning the hash table's keys is very efficient because Ocient utilizes a custom-designed hash table layout specifically optimized for cache locality, ensuring that key iteration is a dense sequential memory access. An additional optimization comes into play for BFs specifically. If the optimizer detects that multiple BFs will be unioned together, it sets a lower probabilistic threshold for discarding node-local BFs. This is done to avoid a unioned filter that is too dense and, thus, has a reduced filtering effect. In general, however, compute resources are used more efficiently overall.

Prefilters are always relayed from the source to the target via the prefilter manager. This allows us to perform operations like unioning or intersecting prefilters in a central place, and keeps track of all metadata. It is not necessary to pass prefilters along the chains of operators, and the passing of prefilters is mostly decoupled from the actual operators.

3.2 Graceful Degradation

To manage memory and performance overhead, the system employs a graceful degradation strategy based on configurable thresholds. By default:

- If the number of unique keys in the global **key list** exceeds 10,000, the list is discarded, and only the BF is used.
- If the fill ratio of the global **BF** exceeds 70%, it is not used and prefiltering is disabled entirely for the query.

This ensures that we pay the cost of prefiltering only when the filters are likely to be highly selective. These thresholds were chosen based on experimental evaluation of various benchmarks and real-world customer workloads. They successfully facilitate a smooth transition between key lists and BFs.

3.3 Filter Application at the I/O Layer

The I/O operator on the probe side is designed to accept a prefilter at any time. Due to the `WEAK_RIGHT` scheduling, it is possible for the I/O operator to have begun scanning segments[2] from disk by the time the prefilter arrives. When the prefilter is received, the I/O operator begins using it during its next scheduling cycle. Since Ocient employs a cooperative multi-tasking model with short scheduler quanta (less than 100 ms), the delay between a filter's arrival and its application to subsequent data segments is negligible.

- If a **key list** is available and a matching secondary index exists in the segment on the join key column, the I/O operator performs an index lookup to retrieve only the required rows.
- If a key list is available but no index exists, the list is loaded into a temporary hash set, and incoming rows are filtered against it.
- Otherwise, if a **BF** is available, it is used to test each row's join key, discarding rows that are definitively not in the build-side set. Any false positive rows are filtered out by the join operator.

3.4 Managing Prefilters in the Optimizer

Ocient's optimizer determines which operators will be able to build a prefilter and on which keys. It decides which operators can benefit from prefilters, and which strategies they should use—for example, whether thread-local, node-local, or global filtering is required. Further, the optimizer decides on performance trade-offs involving *tee* operators. Such operators feed their data to multiple parent operators (cf. Figure 5) so that the same data can be processed on multiple branches of the query plan. These decisions are stored in the query plan and communicated to the execution engine to coordinate prefilter propagation.

Our prefilter manager can send prefilters to operators below a *tee*, as long as the prefilters created above the *tee* are properly unioned or the optimizer detects that a filter will be created on one parent branch and then applied to another parent branch. If all parent branches of a *tee* operator generate distinct filters on the same columns, the manager unions these filters together and the operators below then use the single combined filter. If parent branches of a *tee* generate filters with different filter keys, or some generate filters and others do not, the optimizer may choose to "untee" and clone the subtree below the *tee* so that each filter may separately reach the cloned lower operators.

[2]A segment is the primary data storage unit in Ocient. Other systems may use the terms *page*, *data block*, *file*, or others.

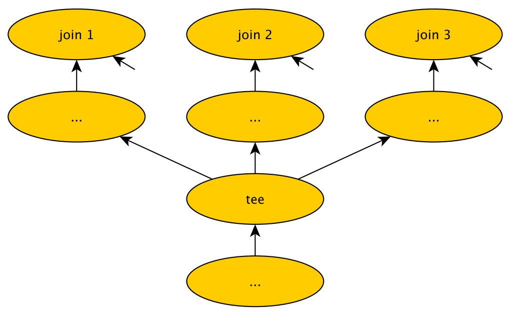


Figure 5: Tee operators

4 Prefiltering for Geospatial Joins

Geospatial joins, such as finding all points within a set of polygons (`ST_Intersects`), are computationally more expensive than simple equi-joins. Ocient's geospatial support is built entirely on geodetic (curved-earth) mathematics, avoiding planar projections. All non-point geometries have a pre-computed bounding box, which is critical for efficient filtering [3, 11].

4.1 Bounding Box List Prefiltering

For geospatial joins where the right side is small enough to broadcast, we apply a prefiltering technique analogous to the equi-join key list. Using the strict `RIGHT` scheduling policy, we first process the entire right side of the join. During this process, we collect the bounding boxes of all geometries. These are aggregated into a **bounding box list**. Eventually, a global bounding box list is created by the same unioning mechanisms that all prefilters use.

This list is then propagated down to the I/O operators on the left (probe) side. If a suitable geospatial index (a Packed Hilbert R-Tree [14]) exists, the list of bounding boxes is used to perform a highly selective index lookup. If no index is present, we discard the filter because we found that scanning with a large in-memory list of bounding boxes is not performant. This limitation is the primary motivation for our future work touched on in Section 6.

4.2 Tiled Execution for Large-Scale Joins

When the right side of a geospatial join is too large to broadcast, the join must fall back to a block nested loop join (BNLJ). This is generally inefficient. To address this, we developed a query rewrite that transforms the spatial join into a highly parallelized hash join based on a dynamic tiling of the Earth's surface.

The core idea is to generate a new `tile_id` column for each geometry. A grid covers the Earth's surface and determines the grid cells (or *tiles*) that the geometry's bounding box intersects, as Figure 6 illustrates. The size of the grid cells is determined by the optimizer based on average bounding box size statistics stored on disk for each geospatial column. By default, the tile dimensions are set to 2× the larger of the average bounding box extents (width and height) from either relation, though this can be overridden via query hints.[3] Since a single geometry can span multiple tiles, this initially involves a one-to-many expansion of rows. The join can then be rewritten as a primary equi-join on `tile_id`, drastically reducing the comparison space.

[3]This is an improvement over static tiling as it was done, e.g., by the IBM DB2 Spatial Extender's grid index mechanism [1, 25], which used three levels of grids. Each grid level had a fixed grid cell size and stored multiple tile ids in a B-tree index.

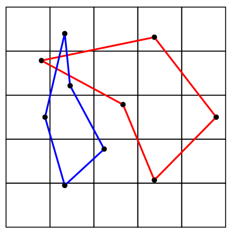


Figure 6: Tiling of geometries

To prevent memory exhaustion, we employ a safety valve for geometries that would cause excessive row expansion. If tiling a geometry would result in a row duplication factor greater than the number of nodes in the cluster (N), we abort the tiling process for that row. Instead, we route the original, untiled geometry to the BNLJ path. In a distributed system, the BNLJ requires broadcasting the build side to all nodes, which effectively incurs a network expansion cost of N. Therefore, any tiling expansion exceeding N would be strictly less efficient than the fallback BNLJ mechanism. Our optimizer automatically rewrites a query with a join predicate like `... ON ST_Intersects(A.geom, B.geom)` into a three-part `UNION` plan, which is depicted in Figure 7:

Main Hash Join: The branch where both `A.geom` and `B.geom` have been successfully tiled. This branch becomes a hash join on the `tile_id` column.

block nested loop join 1: A branch handling every geometry from `A.geom` (including geometries that are too large to be tiled) but only geometries from `B.geom` that were too large. In Ocient's query plans, all joins are either shown as hash joins or product joins. A product join in the plan is implemented via a BNLJ at runtime.

block nested loop join 2: A branch handling cases for tiled geometries from `B.geom` but `A.geom` was too large. All such too-large geometries get a special `tile_id` of −1 assigned.

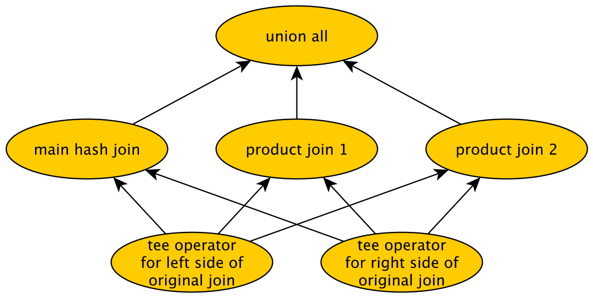

union all
main hash join
product join 1
product join 2
tee operator for left side of original join
tee operator for right side of original join

Figure 7: Geospatial join rewrite

This structure isolates the expensive work on the few oversized geometries, while the vast majority of data flows through the efficient hash join. Note that the three joins are only prefiltering on the bounding boxes. The `ST_Intersects()` predicate must be evaluated on the actual geometries for correct results.

A second challenge is that the initial one-to-many row duplication for tiling leads to duplicate join results. A naive `UNION DISTINCT` is too expensive. We solved this for inner joins with a post-join filtering technique we call *owning tile resolution.*

This resolution happens strictly *before* the expensive geometry computation. The execution sequence is as follows:

(1) **Tile Match:** We join the two relations on the integer `tile_id`. This is a fast equi-join.
(2) **Bounding Box Check:** We verify that the pre-computed bounding boxes of the two geometries actually intersect.
(3) **Owning Tile Check:** If the boxes intersect, we calculate the *intersection rectangle* of the two bounding boxes. We then deterministically select one corner of this intersection rectangle (e.g., the top-left). We calculate which tile ID contains this specific corner. If that tile ID matches the current `tile_id` of the row, we proceed; otherwise, we discard the row. This ensures that a pair of geometries overlapping N tiles is processed exactly once: in the tile that "owns" the intersection corner.
(4) **Precise Geometry Check:** Only if the row passes the owning tile check do we finally execute the expensive `ST_Intersects()` on the actual geometry objects.

This sequence ensures that the computationally heavy exact geometry checks are never performed on duplicate rows generated by the tiling process.

4.3 Integrating Prefiltering with Tiling

The bounding box list prefiltering works seamlessly with this complex tiled execution plan. The build sides of all three join branches generate their respective bounding box lists. These lists are propagated down to a *tee* operator in the plan, which unions them into a single, comprehensive list. This final list is then pushed to the I/O operators on the probe side, providing I/O pruning for all branches of the tiled join.

5 Evaluation

We evaluate the impact of Ocient's dynamic prefiltering framework on realistic, large-scale customer workloads. Our focus is on end-to-end query performance, measured as elapsed time, when executing analytical queries at hyperscale. Prefiltering consistently accelerates queries whose join keys exhibit strong selectivity, while our graceful degradation strategy avoids regressions when filters are not beneficial.

We structure our evaluation around two complementary workload classes. First, we measure the aggregate impact of equi-join prefiltering on industry-standard analytical benchmarks. Second, we conduct an ablation study on geospatial joins using real-world data to isolate the contributions of BB-List prefiltering and Tiled execution.

5.1 Equi-Join Workload

To evaluate equi-join prefiltering (key lists and BFs), we analyzed the TPC-H benchmark at scale 100 TiB and the TPC-DS benchmark at scale 30 TiB.

Setup. These experiments were performed on an Ocient cluster consisting of 8 nodes. Each node has 2 sockets with 96 cores per socket and hyper-threading enabled, totaling 384 logical CPU cores. Main memory is sized at 2 TB, configured with a 2:1 huge-page-to-heap ratio. For storage, every node was provisioned with twelve 7.68 TB NVMe SSDs.

Overhead and Bottlenecks. BF construction generally incurs a 0.5 to 2.0 second overhead. While this is irrelevant for complex analytical queries, it can impact sub-second lookup queries; consequently, prefiltering can be disabled via optimizer hints for such latency-sensitive workloads. Without prefiltering, the bottleneck in hyperscale joins is typically the CPU and network load of the distributed hash join itself, processing a large volume of non-qualifying rows. By pruning data at the source, prefiltering shifts the bottleneck back to the I/O layer. Thanks to CASA, we can efficiently handle this I/O load using index lookups or, in the case of BFs, by scanning only the filter columns before reading the wider row.

Benchmark Workloads. We tested the TPC-H and TPC-DS benchmarks, comparing total runtime with prefiltering enabled versus disabled. Table 1 summarizes these experiments.

Table 1: Total runtime for TPC-H and TPC-DS workloads

Workload	Disabled	Enabled	Speedup
TPC-H, 100 TiB	6,652 s	5,218 s	1.27× (21.5%)
TPC-DS, 30 TiB	11,346 s	8,835 s	1.28× (22.1%)

Significant improvements were observed across both benchmarks. For TPC-H, specific join-heavy queries saw substantial reductions in runtime. For example, Q9 improved from 1,222 s to 613 s (2× speedup), and Q8 improved from 909 s to 258 s (3.5× speedup). Previously, complex star-schema joins often saturated the interconnect between nodes in the cluster; prefiltering now effectively prunes dimension keys at the scan operator, reducing the shuffle volume significantly. Where previously TPC-DS showed mixed results, recent optimizations to the prefilter manager and improved degradation thresholds have resulted in a consistent 1.28× speedup across the entire suite, with no significant regressions.

5.2 Geospatial Join Workload

Setup. These experiments were conducted on a 10-node cluster, with each node deployed on Supermicro SYS-120U-TNR Gen3 servers equipped with Intel Xeon Gold 6348 processors (2.60 GHz, 28 cores). Each node provided 2 TB of system memory configured with a 2:1 huge-page-to-heap ratio and twelve 7.68 TB NVMe SSDs.

Methodology and Workload. To evaluate the interaction between our two primary geospatial optimizations—Bounding Box List (BB-List) prefiltering and the Tiled Hash Join rewrite—we performed an ablation study on a diverse set of queries based on the U.S. Census Bureau's TIGER/Line dataset. This dataset comprises approximately 55 million total geometries across all layers, including address ranges (23.8 million), roads (18.1 million), and hydrography features (7.9 million). The workload was constructed to represent two distinct usage patterns:

Targeted Regional Analysis (Broadcast Path): Queries that join large national datasets (e.g., `rails`, `rivers`) against a filtered subset of features (e.g., `point_lm` or `counties`). To isolate the performance of the BB-List prefilter—which relies on the build side being broadcast to all nodes—we restricted the build side cardinality (limit 9,000) in these tests. This mimics common dashboarding scenarios where users filter a massive dataset against a specific list of regions or assets.

Large Scale Joins (Sharded Path): Queries that involve joining two large relations (millions of rows each) where neither fits in memory, e.g. `point_lm` $\bowtie$ `addr_ranges`. This forces the optimizer to utilize the Tiled Hash Join strategy, testing the system's ability to handle cardinality expansions.

The workload stresses the geometry engine with a full spectrum of predicates, ranging from standard bounding box overlaps (&&) to more computationally expensive relationship checks (`ST_Contains`, `ST_Within`).

Results and Analysis. The overhead of constructing bounding box lists is negligible (statistical noise). Figure 8 illustrates the latency reduction for selective joins. To understand the drivers of this performance, Table 2 details the execution time for representative queries under different optimization configurations. The results demonstrate that the two optimizations address different performance bottlenecks.

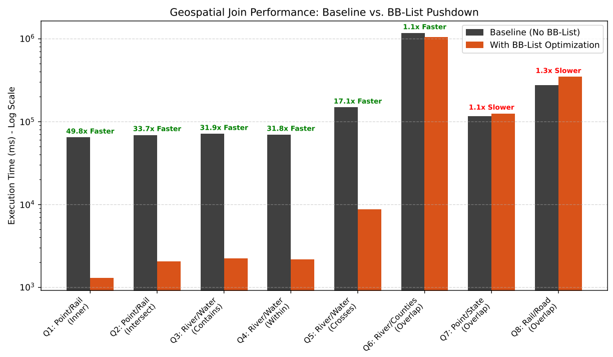


Figure 8: Geospatial Join Performance: Baseline vs. BB-List Pushdown. Note the log scale on the y-axis, highlighting the order-of-magnitude improvements for selective joins.

Scenario 1: Large-Scale Joins (e.g., Micro WL 23). This query joins points with address ranges, resulting in a large candidate set. Here, the *Tiled Hash Join* is the dominant factor, reducing runtime from over 2.7 hours to under 5 minutes (40× speedup). BB-List prefiltering alone fails here; without the tiling rewrite, the engine attempts a BNLJ. The generated bounding box list is too large to trigger an index lookup and is discarded, resulting in a regression compared to baseline due to overhead for prefilter construction.

Scenario 2: High Selectivity (e.g., Pt ⋈ Line). For queries with highly selective predicates or where one side is small (simulated via the Targeted Analysis workload), *BB-List* prefiltering is particularly effective. It reduces runtime by 96% compared to baseline. While Tiling also provides a speedup here (24×), the BB-List approach is slightly faster (2.3 s vs 2.6 s) because it avoids the row-expansion overhead associated with tiling and leverages the spatial index directly at the storage layer.

Scenario 3: Additive Gains. In complex intersection queries (Line ⋈ Line), we observe that the combination of both features yields the best result (1.2 s), outperforming either feature in isolation. The BB-List reduces the I/O pressure on the scan, while the Tiled rewrite efficiently parallelizes the remaining geometry computations.

Conclusion. The combination of Tiled execution and dynamic BB-List prefiltering provides a safety net. Tiling handles the large cardinality expansions that limit traditional geospatial joins, while BB-List prefiltering provides the I/O pruning necessary for high-selectivity lookups. Enabled together, they reduce latency by substantial margins (up to 97% reduction) across a mixed workload without requiring manual tuning.

6 Limitations and Future Work

The primary limitation of our current geospatial prefiltering is the performance cliff experienced when the bounding box list becomes too large to be used with an index. Without an index, the list is discarded, and the I/O operator emits a much larger data volume comprising all rows and geometries. This contrasts with the graceful degradation from a key list to a BF in our equi-join implementation, where some probabilistic filtering is still applied.

To address this, we pursue two complementary approaches. First, a bounding box list can be made smaller by making its entries coarser. The basic idea is to check if there is significant overlap among the bounding boxes and merge the best candidates—at the expense of likely increasing the total area covered. One straightforward way to accomplish that is by Hilbert sorting [12], and then merging nearby bounding boxes if their merged bounding box area does not exceed the original areas by some threshold, say 1.1.

If a bounding box list is still too large after the above optimization, we would like to continue to gracefully degrade. Towards that goal, we are designing a probabilistic two-dimensional data structure, the `GeoFilter`, which will act like a *Geospatial Bloom filter*. This structure maps the globe onto a 2D bit grid and uses an adaptive quadtree-like zoom mechanism to efficiently represent geographically clustered spatial data [20]. Key features of the design include:

Adaptive Zoom: The filter can dynamically "zoom out" to a parent quadtree tile if the data being added is dispersed over a wider area than initially anticipated. This is achieved via an AVX2-accelerated in-place downsampling of the bit grid, allowing the filter to adapt its geographic scope during its construction.

Division-Free Hashing: In the hot path, expensive floating-point divisions are replaced with pre-calculated multiplicative scaling factors, ensuring minimal CPU overhead when testing geometries against the filter.

Saturation Handling: The filter intelligently manages its own density. If it becomes too saturated at a local zoom level, it automatically zooms out. If it becomes saturated at the global level (zoom level 0), it is provably non-selective and discards itself, releasing its memory.

Efficient Bulk Loading: The filter can be initialized from a large list of bounding boxes in a single pass that automatically determines the optimal starting zoom level for the given dataset.

This future implementation will provide a robust, multi-stage filtering pipeline for geospatial joins that mirrors the efficiency and graceful degradation of our existing equi-join system, eliminating the current performance cliff.

In addition to the above, we continue to experiment with scheduler strategies. Currently, `RIGHT` enforces that a prefilter should be generated before the probe side I/O operator instances can start, but this is only a node-local enforcement. Runtime skew across nodes can cause faster nodes to outrun the filters. Further, the scheduler may currently start executing the I/O operator before the prefilter manager has unioned the filters and passed the unioned filter on. A stricter scheduling strategy than `RIGHT` is to force the I/O operator to explicitly await a prefilter before starting to scan the first segment.

Table 2: Ablation study of geospatial optimizations (Time in Seconds)

Query / Scenario	Predicate	Baseline	Spatial Only	BB-List Only	Both	Primary Gain
Micro Workload 23	`ST_Intersects`	9,941 s	243 s	10,146 s	272 s	Tiling
Pt ⋈ Line (Intersect)	`ST_Intersects`	63.7 s	2.6 s	2.3 s	2.3 s	BB-List
Line ⋈ Line (Overlap)	`ST_Overlaps`	6.6 s	1.3 s	2.6 s	1.2 s	Both
Nested Semi-Join	`ST_Contains`	231.4 s	8.7 s	9.2 s	9.2 s	Equiv.

7 Related Work

Many systems employ storage-level metadata and statistics to prune pages, segments, or files before they are read. Snowflake, for example, stores data in micro-partitions that are augmented with rich statistics such as per-column ranges and distinct value summaries [7]. At query time, predicates are evaluated against this metadata to skip entire micro-partitions that cannot satisfy the filter. In contrast, our framework derives prefilters from the build side of a join at runtime and pushes them into storage-adjacent I/O operators, allowing Ocient to prune segments based on query-specific key sets or bounding boxes rather than only static load-time statistics.

DB2 with BLU Acceleration similarly uses synopsis tables and zone-map-like metadata to avoid scanning irrelevant data [8]. BLU computes compressed columnar summaries during table maintenance and exploits them during scans to skip pages and extents whose min/max values fail the query predicates. While this is conceptually related to our I/O-layer pruning, BLU's pruning is still driven by static summaries of individual columns.

The concept of using runtime information to prune data is not new. Systems like Spark, Trino, Impala, and Snowflake implement features often called *runtime filters* or *dynamic partition pruning* [5, 6, 22, 27, 28]. Recent work has explored this in greater depth; for example, Parachute [23] introduces a bi-directional information passing scheme for joins, and Snowflake has published details on their specific pruning mechanics [13]. However, most of these approaches focus on partition-level pruning or compute-side filtering. Ocient's approach differs by leveraging the CASA architecture to push build-side filters directly into storage-adjacent I/O operators, pruning data at the segment level before it enters the primary interconnect.

In distributed systems, algorithms such as Bloomjoin build a BF over the join keys of one relation and ship it to remote sites to selectively avoid sending tuples that cannot participate in the join. Ramesh et al. [19] extend this idea by adapting the BF length to the cardinality of the build-side relation and selecting between several Bloomjoin variants based on database statistics. Their goal is to minimize network traffic in distributed joins, which is also a concern for Ocient, but currently not the primary one.

Other work has applied Bloom-based reduction techniques in large-scale data processing frameworks. Zhang et al. [29] propose an improved Bloom-filter-based approach to distributed joins in MapReduce, showing how to reduce both data transfer and computation by carefully orchestrating BF construction and application across map and reduce phases.

In the realm of spatial data processing, systems often rely on distributed R-Tree joins [4, 17] or traditional Partition Based Spatial-Merge (PBSM) [18]. Microsoft SQL Server [9] also employs a grid-based approach. However, SQL Server typically relies on a static multi-level grid hierarchy defined at index creation time. In contrast, our tiled execution generates the grid dynamically per-query, determining the optimal tile size based on the specific bounding box statistics of the relations involved in the current join. Recent work on adaptive geospatial joins [21] similarly uses hierarchical grids to approximate polygons, but focuses on in-memory SIMD optimizations rather than distributed join rewrites over petabyte-scale storage. As the Ocient core engine is optimized for columnar hash joins rather than distributed tree navigation, our tiling strategy is required to transform spatial problems into integer-based joins that match the engine's native architecture.

Our work takes a different angle and innovates in several key aspects. First, our prefiltering mechanism is deeply intertwined with Ocient's specific push-based execution model and scheduling policies, which create the precise conditions for this optimization. Second, our framework is unified, applying the same core principle of filter generation and propagation to both standard equi-joins as well as complex geospatial joins with the different data structures each requires. Finally, our tiled execution strategy for large geospatial joins, including the "owning tile" technique for duplicate elimination, is a novel approach to transforming these computationally difficult problems into highly parallelizable hash joins that can still benefit from dynamic prefiltering.

8 Conclusion

In this paper, we presented a dynamic prefiltering framework for accelerating massive-scale joins in the Ocient Hyperscale Data Warehouse. By leveraging our push-based execution and join scheduling model, we generate filters (key lists, BFs, and bounding box lists) from the build side of a join at runtime. These filters are propagated to storage-adjacent I/O operators to prune non-matching data from the probe side at the earliest possible stage, reducing I/O, network traffic, and subsequent computational load.

We demonstrated how this unified framework is applied both to high-throughput equi-joins, with a graceful degradation from exact to probabilistic filtering, and to complex geospatial joins. For the latter, we detailed a novel query rewrite strategy that converts intractable spatial joins into efficient, tiled hash joins on simple integers, which are then further accelerated by our prefiltering mechanism. Our evaluation on industry-standard benchmarks shows consistent speedups exceeding 20% for equi-join workloads, while geospatial join latency is reduced by up to 97%. This work showcases how deep integration between query scheduling, execution, and I/O can improve performance for critical analytical workloads at hyperscale.

Trademarks

References

[1] David William Adler and Knut Stolze. 2010. Reducing index size for multi-level grid indexes. https://patents.google.com/patent/US7836082.

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[3] Hermann Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. ACM, 237–246.

[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 237–246. https://doi.org/10.1145/170035.170075

[5] Cloudera. 2025. Runtime Filtering for Impala Queries. https://impala.apache.org/docs/build/html/topics/impala_runtime_filtering.html. [accessed: 2025-11-25].

[6] Ember Crooks. 2024. Stating and Dynamic Partition Pruning in Snowklake. https://datageek.blog/2024/07/16/static-and-dynamic-partition-pruning-in-snowflake. [accessed: 2025-11-27].

[7] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[8] Vijayshankar Raman et al. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* (2013). https://doi.org/10.14778/2536222.2536233

[9] Y. Fang et al. 2008. Spatial Indexing in Microsoft SQL Server 2008. In *SIGMOD*.

[10] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (Jan. 2012), 45–51. https://doi.org/10.1145/2094114.2094126

[11] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. ACM, 47–57.

[12] Yasunobu Imamura, Takeshi Shinohara, Kouichi Hirata, and Tetsuji Kuboyama. 2016. Fast Hilbert Sort Algorithm Without Using Hilbert Indices. In *Similarity Search and Applications - 9th International Conference, SISAP 2016, Tokyo, Japan, October 24-26, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9939)*, Laurent Amsaleg, Michael E. Houle, and Erich Schubert (Eds.). 259–267. https://doi.org/10.1007/978-3-319-46759-7_20

[13] Snowflake Inc. 2025. Pruning at Snowflake: Work Smarter, Not Harder. In *SIGMOD*.

[14] Ibrahim Kamel and Nick Roussopoulos. 1993. On Packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management*. 490–499. https://doi.org/10.1145/170088.170403

[15] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* 6, 1 (2021), 86–101. https://doi.org/10.1007/S41019-020-00149-7

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[17] M. G. Martynov. 1996. Spatial Joins and R-trees. In *Advances in Databases and Information Systems*, Johann Eder and Leonid A. Kalinichenko (Eds.). Springer London, London, 295–304.

[18] Jignesh M. Patel and David J. DeWitt. 1996. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/233269.233338

[19] Sanjay Ramesh, Odysseas Papapetrou, and Wolfram Siberski. 2008. Optimizing Distributed Joins with Bloom Filters. In *Distributed Computing and Internet Technology*, Manish Parashar and S. K. Aggarwal (Eds.). Lecture Notes in Computer Science, Vol. 5375. Springer. https://doi.org/10.1007/978-3-540-89737-8_15

[20] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (1984), 187–260. https://doi.org/10.1145/356924.356930

[21] Stefan Schuh et al. 2018. Adaptive Geospatial Joins for Modern Hardware. *arXiv preprint arXiv:1802.09488* (2018).

[22] Anjali Sharma. 2020. Dynamic Partition Pruning in Spark 3.0. https://dzone.com/articles/dynamic-partition-pruning-in-spark-30. [accessed: 2025-11-27].

[23] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. In *Proceedings of the VLDB Endowment*, Vol. 18.

[24] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013*, Ryan Johnson and Alfons Kemper (Eds.). ACM, 7. https://doi.org/10.1145/2485278.2485285

[25] Knut Stolze, Ying Chen, and Fang Yan Rao. 2008. Systems, methods, and computer program products to reduce computer processing in grid cell size determination for indexing of multidimensional databases. https://patents.google.com/patent/US7437372B2.

[26] Knut Stolze, Dan Zollers, Andrew Park, and Jason Arnold. 2025. The Ocient Hyperscale Data Warehouse: A Compute-Adjacent Architecture for Extreme-Scale Analytics. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik (GI).

[27] Trino Software Foundation. 2025. Trino 478 Documentatino — Dynamic filtering. https://trino.io/docs/current/admin/dynamic-filtering.html. [accessed: 2025-11-25].

[28] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[29] Cheng Zhang, Lei Wu, and Jing Li. 2012. Optimizing Distributed Joins with Bloom Filters Using MapReduce. In *GDC, IESH and CGAG 2012*, Tai hoon Kim, Hojjat Adeli, Osvaldo Gervasi, and Stephen S. Yau (Eds.). Communications in Computer and Information Science, Vol. 351. Springer.